Compressed Text Databases with Efficient Query Algorithms based on the Compressed Suffix Array

Kunihiko Sadakane

Department of System Information Sciences
Graduate School of Information Sciences
Tohoku University
sada@dais.is.tohoku.ac.jp

Abstract. A compressed text database based on the compressed suffix array is proposed. The compressed suffix array of Grossi and Vitter occupies only $O(n)$ bits for a text of length n; however it also uses the text itself that occupies $O(n \log |\Sigma|)$ bits for the alphabet Σ. On the other hand, our data structure does not use the text itself, and supports important operations for text databases: *inverse*, *search* and *decompress*. Our algorithms can find *occ* occurrences of any substring P of the text in $O(|P| \log n + occ \log^{\epsilon} n)$ time and decompress a part of the text of length l in $O(l + \log^{\epsilon} n)$ time for any given $1 \geq \epsilon > 0$. Our data structure occupies only $n(\frac{2}{\epsilon}(\frac{3}{2} + H_0 + 2 \log H_0) + 2 + \frac{4 \log^{\epsilon} n}{\log^{\epsilon} n - 1}) + o(n) + O(|\Sigma| \log |\Sigma|)$ bits where $H_0 \leq \log |\Sigma|$ is the order-0 entropy of the text. We also show the relationship with the opportunistic data structure of Ferragina and Manzini.

1 Introduction

As the cost of disks decreases and the number of machine-readable texts grows, we can use huge amount of disks and store huge amount of text databases in them. Therefore it becomes difficult to find data from the databases and text search techniques become important. Traditional algorithms perform sequential search to find a keyword from a text; however it is not practical for huge databases. We create indices of the text in advance for querying in sublinear time.

In the area of text retrieval, the inverted index [7] is commonly used due to its space requirements and query speed. The inverted index is a kind of *word indices*. It is suitable for English texts, while it is not suitable for Japanese texts or biological sequences because it is difficult to parse them into words. For such texts a kind of *full-text indices* is used, for example suffix arrays [14] and suffix trees [15]. These indices enable finding any substring of a text. However, the size of full-text indices are quite larger than that of word indices. Recent researches are focused on reducing the sizes of full-text indices [13, 6, 5, 10]. The compressed suffix array of Grossi and Vitter [6] reduces the size of the suffix

array from $O(n \log n)$ bits to $O(n)$ bits. It is used with succinct representation of suffix trees in $O(n)$ bits. Though they only considered binary alphabets, it can be generalized for texts with alphabet size $|\Sigma| > 2$. However, this representation also uses the text itself to obtain a part of the text containing a given pattern. The text occupies $n \log |\Sigma|$ bits. We assume that the base of logarithm is two. Indeed, the data size is always larger than that of the original text size.

Though some algorithms for finding words from a compressed text have been proposed [4, 12], the algorithms have to scan the whole compressed text. As a result, their query time is proportional to the size of the compressed text and they are not applicable to huge texts. Though a search index using the suffix array of a compressed text has also been proposed [16], it is difficult to search arbitrary strings because the compression is based on word segmentation of the text. Furthermore, this search index can be also compressed by our algorithm. The opportunistic data structure of Ferragina and Manzini [5] allows to enumerate any pattern P in a compressed text in $(|P| + occ \log^{\epsilon} n)$ time for any fixed $1 \geq \epsilon > 0$. The compression algorithm is the block sorting [2] based on a permutation of the text defined by the suffix array. It has good compression ratio and fast decompressing speed. Space occupancy of the opportunistic data structure of Ferragina and Manzini is $O(nH_k) + O(\frac{n}{\log n}(\log \log n + |\Sigma| \log |\Sigma|))$ bits where H_k is the order-k entropy of the text. Unfortunately, the second term is often too large in practice.

In this paper we propose algorithms for compressed text databases using the compressed suffix array of Grossi and Vitter. We support three basic operations for text databases, *inverse*, *search* and *decompress*, without using the text itself. Since we do not need the original text, the size of our data structure can become smaller than the text size. The *inverse* returns the inverse of the suffix array. It has many applications, for example the lexicographic order of a suffix in any part of the text can be efficiently computed by using the inverse of the suffix array. This enables efficient proximity search [11], which finds positions of keywords which appear in the neighborhood.

The *search* returns the interval in the suffix array that corresponds to a pattern P from the text of length n in $O(|P| \log n)$ time. The *decompress* returns a substring of length l in the compressed database in $O(l + \log^{\epsilon} n)$ time. Space occupancy is only $n(\frac{2}{\epsilon}(\frac{3}{2} + H_0 + 2 \log H_0) + 2 + \frac{4 \log^{\epsilon} n}{\log^{\epsilon} n - 1}) + o(n) + O(|\Sigma| \log |\Sigma|)$ bits where H_0 is the order-0 entropy of the text and $1 \geq \epsilon > 0$ is a fixed constant. Assume that $n < 2^{32}$ and $H_0 = 2$, which is practical for English texts. If we use $\epsilon = 0.8$, $\log^{\epsilon} n \approx 16$. Then the size of the index is approximately $15n$ bits. On the other hand, the text itself and its suffix array occupies $8n + 32n = 40n$ bits. Therefore our search index reduces the space complexity by 63%.

2 Preliminaries

We consider a problem of enumerating all positions of occurrences of a pattern $P[1..m]$ in a text $T[1..n]$. We create in advance a data structure for finding any pattern from T to achieve query time that is sublinear to n. In this paper we call

a data structure with such a query function *search index*. Examples of search indices are the inverted index [7], the q-gram index [9], the suffix array [14] and the suffix tree [15].

2.1 Inverted index

Inverted index [7] is a set of lists of positions in the text T. Each list corresponds to a word and consists of positions of all occurrences of the word. The positions of each word are located in a consecutive regions of memory or disk in sorted order of the positions. The pointer to the head of each list is stored in a table. In the implementation of the data structure, we can compress the size of indices by encoding positions not by the original values but by the difference of positions between adjacent occurrences. The size of the inverted index is $O(n)$ bits in practice. Query time is $O(m + occ)$ where occ is the number of occurrences of P.

2.2 Rank and select algorithms

We briefly describe the rank algorithm of Jacobson [8] and the select algorithm [17, 18] used in both the compressed suffix arrays [6, 5] and in our algorithm. A function $rank(T, i, c)$ returns the number of occurrences of a character c in a prefix of the string $T[1..i]$ of $T[1..n]$. The most basic case is that the alphabet is binary. In this case, the *rank* function can be implemented easily by using table lookups. We use three levels of tables; each table can be accessed in constant time. In the first table, the numbers of ones in prefixes $T[1..i \log^2 n]$ are stored. In the second table the numbers of ones in substrings $T[1 + i \log^2 n + j \log n..i \log^2 n + (j + 1) \log n]$ are stored. The third table stores values of ranks for all strings of length $\frac{\log n}{2}$. This table has size $\sqrt{n} \cdot \log \frac{\log n}{2} \cdot \frac{\log n}{2}$ bits and it can be accessed in constant time. We use this table two times. Therefore the function *rank* takes constant time. It is possible to implement *rank* for non-binary strings in constant time. We call the tables *directory* of the string. Additional space for the directory is $O(n \log \log n / \log n)$ bits.

We also use the inverse of the rank function, *select* function. A function $select(B, i)$ returns the position of i-th one in the bit-vector $B[1..n]$. We group positions of $\log n \log \log n$ ones and use different encoding according to their ranges, difference between positions of the first one and the last one. If the range is too large, we encode the positions by fixed-length encodings. Otherwise we encode them by difference between adjacent ones like the inverted index. Additional bits for the directory is $O(n / \log \log n)$ bits.

2.3 Compressed suffix arrays

Very recently two types of compressed suffix arrays have been proposed [6, 5]. The suffix array [14] of a text $T[1..n]$ is an array $SA[1..n]$ of lexicographically sorted indices of all suffixes $T[j..n]$. An equation $SA[i] = j$ means that the suffix $T[j..n]$ is lexicographically the i-th smallest suffix in T. It can find any substring

P of length m from a text of length n in $O(m \log n + occ)$ time where occ is the number of occurrences of P. It is further reduced to $O(m + \log n + occ)$ time by using an auxiliary array.

A defect of the suffix array is its size. It occupies $O(n \log n)$ bits. To reduce the size, two types of compressed suffix arrays were proposed. The search index of Grossi and Vitter [6] is an abstract data structure which has size $O(n \log \log n)$ bits and supports an access to the i-th element of a suffix array in $O(\log \log n)$ time, whereas in the case of the suffix array it has size $n \log n$ bits and supports constant time access. Thus, it can find positions of P in $O((m \log n + occ) \log \log n)$ time using only the compressed suffix array and the text. This search index is further extended to have $O(n)$ size and support $O(\log^{\epsilon} n)$ time access for any fixed constant $1 \geq \epsilon > 0$.

The compressed suffix array of Grossi and Vitter [6] is a hierarchical data structure. The k-th level stores information on the suffix array of a text of length $n/2^k$ in which a character consists of 2^k consecutive characters in T. The $l = \lceil \log \log n \rceil$-th level stores a suffix array explicitly. To achieve $O(n)$ size, only the 0-th, the $l' = \lceil \frac{1}{2} \log \log n \rceil$-th level and the l-the level are used. Each level occupies $O(n)$ bits; therefore the total of the three levels is also $O(n)$ bits. The $O(\log^{\epsilon} n)$ access time is achieved by using more levels. Fig. 1 shows an example of the compressed suffix array for a text of length 16. The second level is skipped. Suffixes in the level 0 are divided into two groups: even suffixes and odd suffixes. Even suffixes are represented by the corresponding suffix in the level-1 and odd suffixes are represented by the even suffix in the level-0 using a function Ψ. In level-1, a suffix whose index is a multiple of four is represented by the level-3 suffix. The function Ψ is defined by the following formula:

$$SA[\Psi[i]] = SA[i] + 1.$$

The function Ψ is the key of our algorithms.

The element of the hierarchical suffix array $SA_k[i]$ is represented by

$$SA_k[i] = \begin{cases} d \cdot SA_{k+1}[rank(B_k, i, 1)] & (B_k[i] = 1) \\ SA_k[\Psi[i]] & (B_k[i] = 0) \end{cases}$$

where $B_k[i] = 1$ represents that suffix $SA_k[i]$ is also stored in the next level.

2.4 The opportunistic data structure

There is another compressed suffix array called *opportunistic data structures*. The search index of Ferragina and Manzini [5] has size at most $O(nH_k) + o(n)$ bits where H_k is the k-th order empirical entropy of the text T. It can find P in $O(m + occ \log^{\epsilon} n)$ time for any fixed constant $1 \geq \epsilon > 0$. This search index is based on the block sorting compression algorithm [2]. The block sorting compresses a text after permuting characters by the BW-transformation. After the transformation the string becomes more *compressible* and it can be compressed by a simple algorithm. The transformation is defined by the suffix array $SA[1..n]$ of a text $T[1..n]$. We assume that the last character of the text does not appear in other

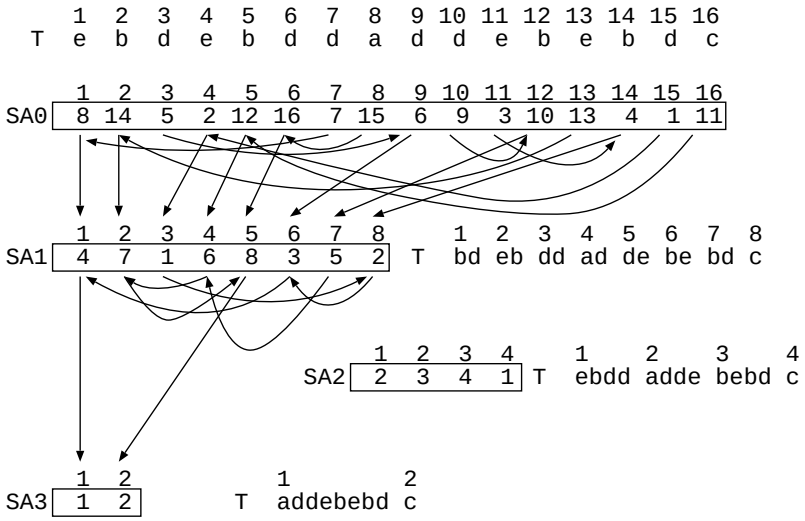
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
T e b d e b d d a d d e b e b d c
SA0 8 14 5 2 12 16 7 15 6 9 3 10 13 4 1 11
SA1 4 7 1 6 8 3 5 2
T bd eb dd ad de be bd c
SA2 2 3 4 1
T ebdd adde bebd c
SA3 1 2
T addebebd c

Fig. 1. The compressed suffix array

positions. Then the result of the BW-transformation is a string $W[1..n]$ where $W[i] = T[SA[i]-1]$, that is, characters in W are arranged in lexicographic order of their following strings. If $SA[i] = 1$, we let $W[i] = T[n]$. Fig. 2 shows an example of the BW-transformation.

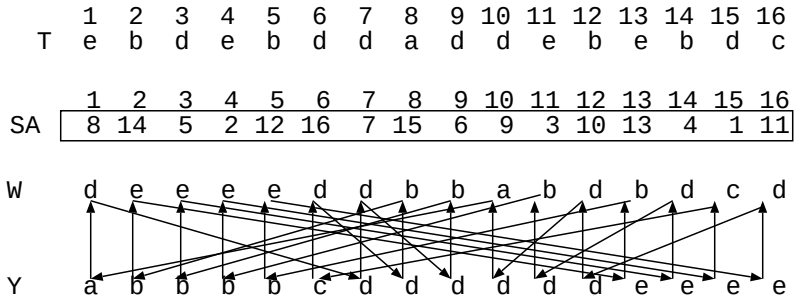

Fig. 2. The BW-transformation

To decompress the compressed string, the inverse of the BW-transformation is computed as follows. First, the string W is converted to a string Y in which characters are arranged alphabetically. Secondly, for each character c, i-th occurrences of c in W and Y are linked by a pointer. Finally the pointers are traversed to recover the original string T. The array Y can be represented by

an integer array C whose size is the same as alphabet size. The entry $C[c]$ is the summation of occurrences of characters that are smaller than c. Then the pointer from $W[i]$ to $Y[j]$ is calculated by

$$j = LF[i] \equiv C[W[i]] + rank(W, i, W[i]).$$

This implies that

$$SA[LF[i]] = SA[i] - 1.$$

Note that Y corresponds to the *first* characters of suffixes and W corresponds to the *last* characters of cyclic shift of T. Hence it is called LF-mapping. It can be calculated in constant time by using the array C and the directory for W.

The string W is compressed by using the recency rank [1] or interval rank. The interval rank of an entry $W[i]$ of W is $i - l(i)$, where $l(i)$ is the position of the rightmost occurrence to the left of i of the same characters as $W[i]$, while the recency rank is the number of distinct characters between $W[i]$ and $W[l(i)]$. Therefore the recency rank of a character is at most the interval rank. The ranks are encoded by the δ-code [3]. The δ-code encodes a number r in $1 + \lfloor \log r \rfloor + 2\lfloor \log(1 + \log r) \rfloor$ bits. Therefore compression ratio using recency ranks is better than using interval ranks. The compression ratio is close to the entropy of the string.

From the compressed text we can recover both the text T and its suffix array SA. Therefore W can be used as a compressed suffix array. It was shown that by traversing the pointers until $SA[1]$, the suffix array SA is recovered in $O(n)$ time [20]. It is necessary to recover only a small part of SA quickly in the context of pattern matching. We logically mark suffixes $T[j..n]$ ($j = 1 + k\eta$, $\eta = \Theta(\log^2 n)$) and store explicitly the index j in a table using its lexicographical order i ($j = SA[i]$) as the key. The mark is stored in Packet B-tree. It occupies $O(n \log\log n / \log n)$ bits. To achieve $O(\log^{\epsilon} n)$ access time, some levels of W are used like the compressed suffix array of Grossi and Vitter.

A problem of the search index of Ferragina and Manzini is its practical space occupancy. It stores a table of recency ranks for each piece of W of length $O(\log n)$. The space occupancy becomes $O(\frac{n}{\log n}|\Sigma| \log |\Sigma|)$ bits. Though it is reduced by using heuristics, it will be large for large alphabets.

3 Text databases using the compressed suffix array

We propose algorithms for compressed text databases using the compressed suffix array of Grossi and Vitter. We support three more operations in addition to the original compressed suffix array:

- *inverse*(j): return the index i such that $SA[i] = j$.
- *search*(P): return an interval $[l, r]$ of indices of SA where substrings $T[SA[i]..SA[i] + |P| - 1]$ ($l \leq i \leq r$) matches with the pattern P.
- *decompress*(s, e): return the substring $T[s..e]$.

In text databases these operations are very important. A database returns texts or sentences containing given keywords [21] using the *search* and the *decompress* functions. The *inverse* function is a fundamental function and it has many applications.

The search index of Ferragina and Manzini is a compressed text by the block sorting with directories for fast operations; therefore it can recover both the original text and its suffix array. On the other hand, the compressed suffix array of Grossi and Vitter is used with the text or the Patricia trie. A binary search on the suffix array requires the text for string comparisons. However, we show that the text is not necessary to perform a binary search. We also show that a part of the text can be decompressed using an additional array.

The data structure of our search index is based on the compressed suffix array. They differ in the Ψ function. In the basic case of the compressed suffix array, the values of the Ψ function are stored only for odd suffixes, while ours stores for all suffixes. We also use auxiliary arrays to compute inverse of the suffix array and directories for the select function.

3.1 The inverse of the suffix array

We show an $O(\sqrt{\log n})$ time algorithm for the inverse function $i = SA^{-1}[j]$. It is modified to $O(\log^{\epsilon} n)$ time for any given $1 \geq \epsilon > 0$ by using the same technique as the compressed suffix array [6].

We use three levels of the compressed suffix array: 0, $l' = \lceil \frac{1}{2} \log\log n \rceil$ and $l = \lceil \log\log n \rceil$. In the l-th level the suffix array SA_l of size $n/\log n$ is explicitly stored. We also store its inverse SA_l^{-1} explicitly in the l-th level. To find $SA^{-1}[i + j\sqrt{\log n} + k \log n]$ becomes as follows. First, we obtain $x_1 = SA_l^{-1}[k]$. This corresponds to $x_2 = SA_{l'}^{-1}[k\sqrt{\log n}]$, which is calculated by $x_2 = \mathit{select}(B_{l'}, x_1)$. Secondly, we obtain $x_3 = SA_{l'}^{-1}[j + k\sqrt{\log n}]$ by $x_3 = (\Psi')^j[x_2]$ because $SA_{l'}[x_3] = j + k\sqrt{\log n} = SA_{l'}[x_2] + j = SA_{l'}[(\Psi')^j[x_2]]$. Thirdly, we obtain $x_4 = SA^{-1}[j\sqrt{\log n} + k \log n]$ by $x_4 = \mathit{select}(B_0, x_3)$. Finally, we obtain $x = SA^{-1}[i + j\sqrt{\log n} + k \log n]$ by $x = (\Psi)^i[x_4]$.

To achieve the $O(\log^{\epsilon} n)$ time complexity, we use $1/\epsilon$ levels. In each level the Ψ function is used at most $\log^{\epsilon} n$ times. Thus we have this theorem.

Theorem 1. *The inverse of the suffix array $SA^{-1}[j]$ can be computed in $O(\log^{\epsilon} n)$ time.*

3.2 Searching for pattern positions

We propose an algorithm for finding the interval $[l, r]$ in the suffix array that corresponds to suffixes containing a given pattern P. Our algorithm runs in $O(|P| \log n)$ time, which is the same as the original suffix array. Our algorithm has two merits. One is that the text itself is not necessary for string comparisons and the other is that in the time complexity a multiplicative factor $\log^{\epsilon} n$ does not appear even if the compressed suffix array is used.

We search for a pattern P by a binary search on the suffix array $SA[1..n]$. In a classical algorithm P is compared with the suffix $T[SA[m]..n]$. Therefore $O(\log^\epsilon n)$ time is necessary to find $SA[m]$ and the text T is used to obtain the suffix. On the other hand, our algorithm decodes the suffix from the compressed suffix array. We use the following proposition.

Proposition 1.

$$T[SA[m] + i] = C^{-1}[\Psi^i[m]]$$

The function $C^{-1}[j]$ is the inverse of the array of cumulative frequencies C, that is, it returns the character $T[SA[j]]$.

Proof. Assume that $SA[m] = j$. Note that we do not know the exact value of j. Since suffixes are lexicographically sorted in the suffix array, The first character $T[j]$ of the suffix $T[j..n]$ becomes $C^{-1}[m]$. To compute the i-th character $T[j+i]$, we need the lexicographical position m' such that $SA[m'] = j + i$. By using the relation $SA[\Psi[m]] = SA[m] + 1$, we can calculate $m' = \Psi^i[m]$. □

The function C^{-1} takes constant time using a bit-vector $D[1..n]$ and the directory for the rank function. We set $D[j] = 1$ if $T[SA[j]] \neq T[SA[j-1]]$. Then $rank(D, j, 1)$ represents the number of distinct characters in $T[SA[1]]$, $T[SA[2]]$, $\ldots, T[SA[j]]$ minus one. The rank can be converted to the character $T[SA[j]]$ by a table of size $|\Sigma| \log |\Sigma|$ bits.

By using the proposition, we can obtain the substring of T used in a string comparison in $O(|P|)$ time, which is the same as using the uncompressed suffix array and the text itself.

3.3 Decompressing the text

We show an algorithm to obtain a substring of length l, $T[j..j+l-1]$, from the text of length n in $O(l+\log^\epsilon n)$ time. It can be performed by using the inverse of the suffix array and Proposition 1. We can compute the lexicographical position i of the suffix $T[j..n]$ ($i = SA^{-1}[j]$) in $O(\log^\epsilon n)$ time. After that we can obtain the prefix $T[j..j+l-1]$ of the suffix in $O(l)$ time.

Theorem 2. *A substring of length l can be decompressed from the compressed suffix array of a text of length n in $O(l + \log^\epsilon n)$ time.*

Fig. 3 shows an example of decompressing $T[9..13] =$ ddebe. We first calculate the index of the suffix array storing the suffix $T[9..16]$ by using the inverse function $SA^{-1}[9] = 10$. Then we traverse elements of the suffix array by using the Ψ function and decode the characters by using C^{-1}. Note that exact values of $SA^{-1}[i]$ are not necessary except for $SA^{-1}[9]$.

We can also create a small suffix array corresponding a part of the text, $T[s..s+l-1]$, by using the inverse function $SA^{-1}[s]$ and traversing the Ψ function.

Theorem 3. *The suffix array of a part of the text of length l can be computed from the compressed suffix array in $O(l + \log^\epsilon n)$ time.*

Note that it is necessary to sort positions j of suffixes in order of $SA^{-1}[j]$. However it can be done in linear time using a radix sort.

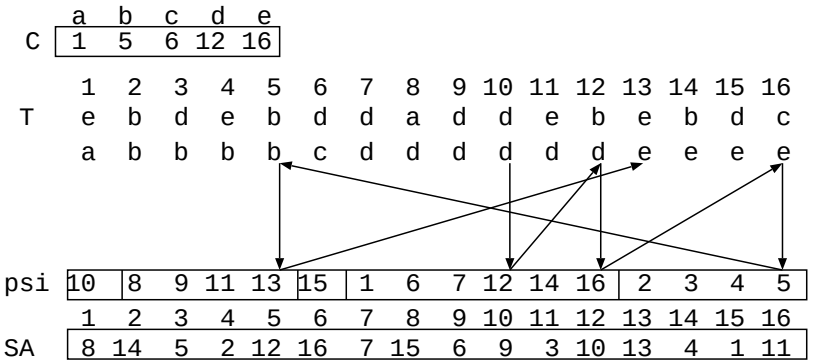

Fig. 3. Decompressing the text

4 The size of the search index

We consider the size of our compressed suffix array. It consists of at most $l+1$ levels ($l = \lceil \log \log n \rceil$). The l-th level is the suffix array SA_l and its inverse SA_l^{-1}. The size of each array is $n \log n / \log n = n$ bits.

In the level k, we store three components: a bit-vector B_k, the function Ψ_k, C_k^{-1}, and their directories. The bit-vector B_k has length $n/2^k$ representing whether each suffix in level k appears in the next level. We use the directory for the rank function. The number of bits for the vector and the directory is $n/2^k + o(n/2^k)$ bits. We also use the select function for the inverse function. It occupies at most $n/2^k + o(n/2^k)$ bits.

For storing Ψ_k, we use the inverted index. In level k, consecutive 2^k characters in the text T form a new character. Therefore the length of the text becomes $\frac{n}{2^k}$. We call the text as T_k and the corresponding suffix array as SA_k. $\Psi_k[i]$ is encoded in a list by the δ-code representing $\Psi_k[i] - \Psi_k[i-1]$ if $T_k[SA_k[i]] = T_k[SA_k[i-1]]$, otherwise encoded by the δ-code of $\Psi_k[i]$.

We represent the size of the Ψ_k function by the entropy of the text. We assume that the probability of a character $c \in \Sigma^{2^k}$ is p_c and that the number of occurrences of c is $n_c = \frac{n}{2^k} p_c$. Then the total number of bits z_k to encode all values of Ψ in level k becomes

$$
\begin{aligned}
z_k &\leq \sum_{c \in \Sigma^{2^k}} n_c \left(1 + \log \frac{n/2^k}{n_c} + 2 \log \log \frac{n/2^k}{n_c} \right) \\
&= \frac{n}{2^k} \sum_{c \in \Sigma^{2^k}} p_c \left(1 + \log \frac{1}{p_c} + 2 \log \log \frac{1}{p_c} \right) \\
&\leq \frac{n}{2^k} (1 + H_{2^k} + 2 \log H_{2^k}) \leq \frac{n}{2^k} (1 + 2^k H_0 + 2 \log 2^k H_0) \\
&= n \left(\frac{2k+1}{2^k} + H_0 + 2 \log H_0 \right) \leq n \left(\frac{3}{2} + H_0 + 2 \log H_0 \right)
\end{aligned}
$$

where H_{2^k} is the order-2^k entropy.

The function C_k^{-1} is used to obtain $\Psi_k[i]$. C_k is represented by a bit-vector $D_k[1..n/2^k]$ for each level k. The element $D_k[i]$ represents whether $T_k[SA_k[i]] = T_k[SA_k[i-1]]$ or not. In other words, i is the first element in a list encoding the Ψ_k values for a character c if $D_k[i] = 1$ and the rank of $D_k[i]$ is c. Therefore we can find the character c using D_k and its directory for the rank function. We also encode positions of ones in D_k and use the directory for the select function to calculate the number of elements $i - \mathit{select}(D_k, \mathit{rank}(D_k, i, 1))$ in the list of Ψ_k corresponding to c. The number of required bits is at most $n/2^k + o(n/2^k)$ bits.

We also use a bit-vector $Z_k[1..z_k]$ to store the bit-position of the first element of a list of Ψ_k for each character as is used in the opportunistic data structure. We make the bit-vector Z_k so that $Z_k[i] = 1$ represents that the position i is the first bit representing the first element of the Ψ_k list. Therefore c-th 1 in Z_k represents the bit-position for the character c. We can compute the bit-position for the character c in constant time using the directory for the select function.

The bit-vectors B_k and D_k and their directories are expressed in $4n/2^k + z_k + o(n)$ bits for the level k. We use levels $0,\ \epsilon \log\log n,\ 2\epsilon \log\log n, \ldots, \log\log n$. Therefore the total of all levels is less than $4n\frac{\log^\epsilon n}{\log^\epsilon n - 1} + \sum_k z_k + o(n)$ bits. Note that it is not necessary to know exact values of C^{-1} except for level 0. We use an array to convert the rank of a character in the text to the character code. Its size is at most $|\Sigma| \log |\Sigma|$ bits.

We use $1/\epsilon$ levels of the compressed suffix arrays to achieve $O(\log^\epsilon n)$ access time. In the last level l, we store two arrays SA_l and SA_l^{-1} explicitly. By adding the above numbers, we have the following theorem:

Theorem 4. *The number of bits of the proposed search index for a text of length n is at most $n(\frac{2}{\epsilon}(\frac{3}{2} + H_0 + 2\log H_0) + 2 + \frac{4\log^\epsilon n}{\log^\epsilon n - 1}) + o(n) + O(|\Sigma| \log |\Sigma|)$ bits where H_0 is the order-0 entropy of the text and $|\Sigma|$ is the alphabet size.*

Note that we encode Ψ explicitly if it is smaller than encoding Ψ and C^{-1} by the above method.

Our search index stores all $n/2^k$ values of Ψ_k in all levels, while the original compressed suffix array stores $\frac{n}{2^k}\frac{\log^\epsilon n - 1}{\log^\epsilon n}$ values of Ψ_k. We also requires additional n bits to store the inverse suffix array SA^{-1} in level l explicitly. However, ours does not require the text itself that occupies $n \log |\Sigma|$ bits.

5 Relationship between two compressed suffix arrays

In this section we show relationship between the compressed suffix array and the opportunistic data structure. Both search indices use pointers between two adjacent suffixes in the suffix array; however they differ in their directions:

$$SA[LF[i]] = SA[i] - 1$$
$$SA[\Psi[i]] = SA[i] + 1.$$

This implies that $LF[\Psi[i]] = i$. Both functions LF and Ψ can be used to decompress the compressed text.

The LF-mapping is encoded by the block sorting compression, while the Ψ function is encoded by the inverted index. Compression ratio of the block sorting is better than the inverted index. However the size of the directory for the rank function highly depends on the alphabet size.

The search indices differ in keyword search time. The compressed suffix array takes $O(|P| \log n)$ time for finding a pattern P if no additional index is used, while the opportunistic data structure takes only $O(|P|)$ time using a technique like the radix sort. Therefore reducing its practical size is important.

Concerning keyword search, we want to perform case-insensitive search. On the other hand, we want to extract the text as it is. We create search indices of a text in which capital letters are converted to the corresponding small letters. We call this conversion *unification*. The modified Burrows-Wheeler transformation [19] allows to decode both the original text before unification and the suffix array of the text after unification. This transformation is also used in the opportunistic data structure. We encode recency ranks of non-unified characters, while directories are computed using unified characters. In the case of using the compressed suffix array, we first create the unified text and then create the index. We have to store flags separately which represents whether each letter is capital or small.

6 Concluding remarks

We have proposed algorithms for text databases using the compressed suffix array. We support keyword search and text decompression. Since our search index does not require the text itself, its size is small even in non-binary alphabet cases. We also support a basic operation which has many applications, inverse of the suffix array. As future works we develop other basic operations which are used to simulate the suffix tree.

Acknowledgment

The author would like to thank Prof. Takeshi Tokuyama and anonymous referees for their valuable comments. The author also wish to thank Dr. Giovanni Manzini, who kindly supplied his paper.

References

1. J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM*, 29(4):320–330, April 1986.
2. M. Burrows and D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithms. Technical Report 124, Digital SRC Research Report, 1994.
3. P. Elias. Universal codeword sets and representation of the integers. *IEEE Trans. Inform. Theory*, IT-21(2):194–203, March 1975.

4. M. Farach and T. Thorup. String-matching in Lempel-Ziv Compressed Strings. In *27th ACM Symposium on Theory of Computing*, pages 703–713, 1995.
5. P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. Technical Report TR00-03, Dipartimento di Informatica, Università di Pisa, March 2000.
6. R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000. `http://www.cs.duke.edu/~jsv/Papers/catalog/node68.html`.
7. D. A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics*. Kluwer Academic Publishers, 1998.
8. G. Jacobson. Space-efficient Static Trees and Graphs. In *30th IEEE Symp. on Foundations of Computer Science*, pages 549–554, 1989.
9. P. Jokinen and E. Ukkonen. Two Algorithms for Approximate String Matching in Static Texts. In A. Tarlecki, editor, *Proceedings of Mathematical Foundations of Computer Science*, LNCS 520, pages 240–248, 1991.
10. J. Kärkkäinen and E. Sutinen. Lempel-Ziv Index for q-Grams. *Algorithmica*, 21(1):137–154, 1998.
11. T. Kasai, H. Arimura, R. Fujino, and S. Arikawa. Text data mining based on optimal pattern discovery – towards a scalable data mining system for large text databases –. In *Summer DB Workshop*, SIGDBS-116-20, pages 151–156. IPSJ, July 1998. (in Japanese).
12. T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A Unifying Framework for Compressed Pattern Matching. In *Proc. IEEE String Processing and Information Retrieval Symposium (SPIRE'99)*, pages 89–96, September 1999.
13. S. Kurtz. Reducing the Space Requirement of Suffix Trees. Technical Report 98-03, Technische Fakultät der Universität Bielefeld, Abteilung Informationstechnik, 1998.
14. U. Manber and G. Myers. Suffix arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.
15. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(12):262–272, 1976.
16. E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In *Proc. of WSP'97*, pages 95–111. Carleton University Press, 1997.
17. J. I. Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Computer Science (FSTTCS '96)*, LNCS 1180, pages 37–42, 1996.
18. J. I. Munro. Personal communication, July 2000.
19. K. Sadakane. A Modified Burrows-Wheeler Transformation for Case-insensitive Search with Application to Suffix Array Compression. In *Proceedings of IEEE Data Compression Conference (DCC'99)*, page 548, 1999. poster session.
20. K. Sadakane and H. Imai. A Cooperative Distributed Text Database Management Method Unifying Search and Compression Based on the Burrows-Wheeler Transformation. In *Advances in Database Technologies*, number 1552 in LNCS, pages 434–445, 1999.
21. K. Sadakane and H. Imai. Text Retrieval by using k-word Proximity Search. In *Proceedings of International Symposium on Database Applications in Non-Traditional Environments (DANTE'99)*, pages 23–28. Research Project on Advanced Databases, 1999.